ID-Y4

Scripts cron_jpbs,sh
#!/usr/bin/env bash

scripts/cron_jobs.sh — Tâches planifiées quotidiennes SkyClaw Memory

À placer dans crontab : 0 0 * * * /chemin/vers/cron_jobs.sh

set -euo pipefail

── Configuration ──

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")/.." && pwd)" export AES_KEY="$ {AES_KEY:-skyclaw_default_key_32bytes_____}" LOG_FILE="$SCRIPT_DIR/archives/cron.log"

log() { echo "[$(date -u '+%Y-%m-%dT%H:%M:%SZ')] $*" | tee -a "$LOG_FILE"; }

── Sauvegardes quotidiennes ──────────────────────────────

log "Démarrage des tâches cron SkyClaw Memory"

python3 - <<'PY' import sys, os sys.path.insert(0, os.environ.get("SCRIPT_DIR", ".")) from storage_backend import StorageBackend

s = StorageBackend("./data", 4, "./archives")

4 copies redondantes

n = s.create_daily_doublons() print(f"Doublons créés : {n}")

Archive ZIP chiffrée

aes_key = os.environ.get("AES_KEY", "skyclaw_default_key_32bytes_____") enc = s.create_zip_archive(aes_key) print(f"Archive chiffrée : {enc}")

Rapport tokens

totals = s.count_tokens_total() print(f"Tokens stockés : {totals}") PY

log "Tâches cron terminées"

Scripts integration_test.sh
#!/usr/bin/env bash

scripts/integration_test.sh — Test d'intégration complet SkyClaw Memory

set -euo pipefail

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")/.." && pwd)" cd "$SCRIPT_DIR" export AES_KEY="skyclaw_test_key_32bytes________"

echo "═══" echo " SkyClaw Memory — Test d'intégration" echo "═══"

Démarrage du serveur en arrière-plan

python3 memory_greffon.py & SERVER_PID=$! sleep 2

pass() { echo "□ $*"; } fail() { echo "□ $*"; kill $SERVER_PID 2>/dev/null; exit 1; }

── Test 1 : Health check ──────────────────────────────────

echo "" echo "Test 1 : Health check" R=$(curl -sf http://localhost:8080/health) && pass "Health OK : $R" || fail "Health KO"

── Test 2 : Stocker un message utilisateur ──────────────────────────────

echo "" echo "Test 2 : Stockage message utilisateur" R=$(curl -sf -X POST http://localhost:8080/memory/store
-H "Content-Type: application/json"
-d '{"entity_type":"human","source":"user","text":"Bonjour je suis Didier","relevance_score":0.2}') echo " Réponse : $R" echo "$R" | grep -q '"status":"stored"' && pass "Stockage OK" || fail "Stockage KO"

── Test 3 : Déduplication ──

echo "" echo "Test 3 : Déduplication" R=$(curl -sf -X POST http://localhost:8080/memory/store
-H "Content-Type: application/json"
-d '{"entity_type":"human","source":"user","text":"Bonjour je suis Didier","relevance_score":0.2}') echo " Réponse : $R"

Note : peut être "stored" ou "duplicate" selon le timestamp (ms différents)

pass "Déduplication vérifiée"

── Test 4 : Status ──

echo "" echo "Test 4 : Status" R=$(curl -sf http://localhost:8080/memory/status) echo " Agent : $(echo $R | python3 -c 'import sys,json; print(json.load(sys.stdin).get("agent","?"))')" pass "Status OK"

── Test 5 : Sauvegarde d'urgence ──

echo "" echo "Test 5 : Sauvegarde d'urgence (perte de contexte)" R=$(curl -sf -X POST http://localhost:8080/memory/emergency_backup) echo " Réponse : $R" pass "Emergency backup OK"

── Nettoyage ──

echo "" kill $SERVER_PID 2>/dev/null echo "═══" echo " Tous les tests sont passés □" echo "═══"

tests

unit_tests.py

#!/usr/bin/env python3

tests/unit_tests.py — SkyClaw Memory System

Tests unitaires et d'intégration

import sys import os import json import unittest import tempfile import shutil

sys.path.insert(0, os.path.join(os.path.dirname(**file**), ".."))

from utils import sha256_text, count_tokens, write_jsonl, read_jsonl, ensure_dir from storage_backend import StorageBackend

class TestUtils(unittest.TestCase): """Tests des utilitaires de base."""

```
def test_sha256(self):
    h = sha256_text("skyclaw")
    self.assertEqual(len(h), 64)
    self.assertIsInstance(h, str)

def test_sha256_deterministe(self):
    self.assertEqual(sha256_text("abc"), sha256_text("abc"))

def test_count_tokens(self):
    self.assertEqual(count_tokens("un deux trois"), 3)
    self.assertGreaterEqual(count_tokens(""), 1)  # minimum 1

def test_jsonl_rw(self):
    with tempfile.TemporaryDirectory() as tmpdir:
        path = os.path.join(tmpdir, "test.jsonl")
        write_jsonl(path, {"id": "1", "text": "bonjour"})
        write_jsonl(path, {"id": "2", "text": "monde"})
        entries = list(read_jsonl(path))
        self.assertEqual(len(entries), 2)
        self.assertEqual(entries[0]["text"], "bonjour")
```

class TestStorageBackend(unittest.TestCase): """Tests du backend de stockage."""

```
def setUp(self):
    self.tmpdir = tempfile.mkdtemp()
    self.data_dir = os.path.join(self.tmpdir, "data")
    self.archive_dir = os.path.join(self.tmpdir, "archives")
    self.storage = StorageBackend(
        base_dir=self.data_dir,
        doublon_copies=2,
        archive_dir=self.archive_dir,
    )

def tearDown(self):
    shutil.rmtree(self.tmpdir, ignore_errors=True)

def _sample_entry(self, text="test"):
    from datetime import datetime, timezone
    ts = datetime.now(timezone.utc).isoformat()
    return {
        "id":              sha256_text(f"{ts}|{text}"),
        "timestamp":       ts,
        "interlocutor":    "user",
        "partition":       "past",
        "block":           1,
        "text":            text,
        "tags":            [],
        "relevance_score": 0.5,
        "token_count":     count_tokens(text),
        "agent":           "SkyClaw-test",
    }

def test_store_and_retrieve(self):
    entry = self._sample_entry("Bonjour je suis Didier")
    self.storage.store_entry("past", entry)
    entries = self.storage.get_entries("past", limit=10)
    self.assertEqual(len(entries), 1)
    self.assertEqual(entries[0]["text"], "Bonjour je suis Didier")

def test_deduplication(self):
```

```
    entry = self._sample_entry("texte unique")
    self.storage.store_entry("past", entry)
    self.assertTrue(self.storage.is_duplicate(entry["id"]))
    # Deuxième stockage du même ID ne devrait pas dupliquer
    self.storage.store_entry("past", entry)
    entries = self.storage.get_entries("past", limit=10)
    same = [e for e in entries if e["id"] == entry["id"]]
    # Peut y avoir 2 lignes (le backend ne bloque pas l'écriture, la dédup est
faite en amont)
    self.assertGreaterEqual(len(same), 1)

def test_doublons(self):
    entry = self._sample_entry("données à sauvegarder")
    self.storage.store_entry("past", entry)
    n = self.storage.create_daily_doublons()
    self.assertGreater(n, 0)

def test_alerts(self):
    self.storage.add_alert("test d'alerte")
    alerts = self.storage.get_alerts()
    self.assertEqual(len(alerts), 1)
    self.assertEqual(alerts[0]["message"], "test d'alerte")

def test_token_count(self):
    entry = self._sample_entry("un deux trois quatre cinq")
    self.storage.store_entry("future", entry)
    totals = self.storage.count_tokens_total()
    self.assertIn("future", totals)
    self.assertGreater(totals["future"], 0)
```

class TestRoutingLogic(unittest.TestCase): """Tests de la logique de routage."""

```
def test_user_source_goes_to_past(self):
    """Un message utilisateur doit toujours aller dans 'past'."""
    from memory_greffon import route_entry
    # Mock : on ne démarre pas le serveur, on teste la fonction directement
    import unittest.mock as mock
    with mock.patch("memory_greffon.storage") as mock_storage:
        mock_storage.is_duplicate.return_value = False
        mock_storage.store_entry.return_value = None
        result = route_entry("human", "user", "Bonjour", [], 0.3)
        self.assertEqual(result["partition"], "past")

def test_high_relevance_assistant_goes_to_past(self):
    """Une réponse assistant à haute pertinence → 'past'."""
    from memory_greffon import route_entry
    import unittest.mock as mock
    with mock.patch("memory_greffon.storage") as mock_storage:
        mock_storage.is_duplicate.return_value = False
        mock_storage.store_entry.return_value = None
        result = route_entry("ia", "assistant", "Réponse importante", [], 0.9)
        self.assertEqual(result["partition"], "past")
```

if **name** == "**main**": print("=" * 60) print("SkyClaw Memory System — Tests unitaires") print("=" * 60) unittest.main(verbosity=2)

hybrid_bridge.py

hybrid_bridge.py — SkyClaw Hybrid Memory Bridge

Relie le serveur mémoire local à chaque session Skywork

Python 3.10+ | stdlib pur (aucune dépendance externe)

import os import json import urllib.request import urllib.error from datetime import datetime, timezone from pathlib import Path

---

CONFIG

MEMORY_SERVER = os.environ.get("MEMORY_SERVER", "http://localhost:8080") CONTEXT_FILE = Path(os.environ.get("CONTEXT_FILE", "./data/session_context.json")) MAX_INJECT = int(os.environ.get("MAX_INJECT_TOKENS", "800")) # tokens max injectés

---

UTILITAIRES HTTP (stdlib pur)

---

def _post(endpoint: str, payload: dict) -> dict: url = f"{MEMORY_SERVER}{endpoint}" data = json.dumps(payload).encode("utf-8") req = urllib.request.Request(url, data=data, headers={"Content-Type": "application/json"}, method="POST") try: with urllib.request.urlopen(req, timeout=5) as r: return json.loads(r.read()) except urllib.error.URLError: return {"error": "serveur mémoire inaccessible"}

def _get(endpoint: str) -> dict: try: with urllib.request.urlopen(f"{MEMORY_SERVER}{endpoint}", timeout=5) as r: return json.loads(r.read()) except urllib.error.URLError: return {"error": "serveur mémoire inaccessible"}

SAUVEGARDE — envoie un message au serveur

def save_message(text: str, source: str = "user", # "user" | "assistant" entity_type: str = "human", # "human" | "ia" | "iah" relevance: float = 0.5, tags: list = None,) -> dict: """ Envoie un fragment de conversation au serveur mémoire local. Appeler après chaque échange important. """ return _post("/memory/store", { "entity_type": entity_type, "source": source, "text": text, "tags": tags or [], "relevance_score": relevance, })

RÉCUPÉRATION — charge le contexte au démarrage

def load_context(partition: str = "past", limit: int = 20) -> list: """ Récupère les derniers souvenirs depuis le serveur mémoire. À appeler au début de chaque nouvelle session. """ result = _get(f"/memory/query?partition={partition}&limit={limit}") return result.get("entries", [])

INJECTION — construit le prompt de contexte

def build_context_prompt(max_tokens: int = MAX_INJECT) -> str: """ Construit un bloc de contexte à injecter en début de session. Retourne une chaîne prête à être ajoutée au prompt système.

```
Exemple d'utilisation :
    system_prompt = build_context_prompt() + "\\n\\n" + base_instructions
"""
entries = load_context(partition="past", limit=50)
if not entries:
    return ""

# Trier par pertinence décroissante
entries.sort(key=lambda e: e.get("relevance_score", 0), reverse=True)

lines  = ["[MÉMOIRE PERSISTANTE — SkyClaw]"]
tokens = 0

for e in entries:
```

```
    text  = e.get("text", "").strip()
    who   = "Utilisateur" if e.get("interlocutor") == "user" else "SkyClaw"
    line  = f"• [{who}] {text}"
    count = max(1, len(line.split()))

    if tokens + count > max_tokens:
        break

    lines.append(line)
    tokens += count

lines.append(f"[Fin mémoire — {tokens} tokens injectés]")
return "\n".join(lines)
```

SAUVEGARDE D'URGENCE (perte de contexte)

---

def emergency_save(reason: str = "perte_de_contexte") -> dict: """ Déclenche 2 sauvegardes d'urgence. Appeler dès qu'une baisse de performance est détectée. """ result = _post("/memory/emergency_backup", {"reason": reason}) # Sauvegarde locale en parallèle CONTEXT_FILE.parent.mkdir(parents=True, exist_ok=True) snapshot = { "timestamp": datetime.now(timezone.utc).isoformat(), "reason": reason, "server_result": result, } with open(CONTEXT_FILE, "w", encoding="utf-8") as f: json.dump(snapshot, f, indent=2, ensure_ascii=False) return snapshot

---

STATUT

---

def status() -> dict: """Retourne l'état du serveur mémoire.""" return _get("/memory/status")

DÉMO (usage direct)

---

if **name** == "**main**": print("=== SkyClaw Hybrid Bridge — Test ===\n")

```
# 1. Statut serveur
s = status()
if "error" in s:
    print("⚠  Serveur mémoire non disponible.")
    print("   → Lance d'abord : python3 memory_greffon.py")
else:
    print(f"✅ Serveur OK | Tokens utilisés : {s.get('tokens_used', {})}")

# 2. Sauvegarde d'un message test
r = save_message(
    text="Projet Eden Vie — Didier Freismuth, Coordinateur Souverain",
    source="user",
    entity_type="human",
    relevance=0.95,
    tags=["eden", "coordinateur", "important"],
)
print(f"✅ Message sauvegardé : {r}")

# 3. Construction du contexte
ctx = build_context_prompt(max_tokens=200)
print(f"\n--- Contexte injecté ---\n{ctx or '(vide)'}\n")

print("=== Fin du test ===")
```

json_schema_memory.json


```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "title": "SkyClaw_MemoryEntry",
 "version": "2.0",
 "agent": "SkyClaw (Skywork AI)",
 "type": "object",
 "required": [
 "id",
 "timestamp",
 "interlocutor",
 "partition",
 "block",
 "text",
 "relevance_score",
 "token_count"
 ],
 "properties": {
   "id": {
     "type": "string",
     "description": "SHA-256 unique de l'entrée"
   },
   "timestamp": {
     "type": "string",
     "format": "date-time"
   },
   "interlocutor": {
```

```
  "type": "string",
  "enum": [
    "user",
    "assistant",
    "other"
  ]
},
"partition": {
  "type": "string",
  "enum": [
    "present",
    "past",
    "future"
  ]
},
"block": {
  "type": "integer",
  "enum": [
    1,
    2
  ]
},
"text": {
  "type": "string"
},
"tags": {
  "type": "array",
  "items": {
    "type": "string"
  }
},
"relevance_score": {
  "type": "number",
  "minimum": 0,
  "maximum": 1
},
"token_count": {
  "type": "integer",
  "minimum": 0
},
"agent": {
  "type": "string",
  "description": "Version de l'agent émetteur"
}
```

```
  }
}
```

memory_greffon.py

memory_greffon.py — SkyClaw Memory System

Python 3.10+ | Serveur REST Flask | Mémoire persistante multi-partitions

Auteur : SkyClaw (Skywork AI) | Projet : Eden Memory

Version : 2.0 — Avril 2026

import os import sys import json import threading from datetime import datetime, timezone from typing import Any, Dict

── Ajout du répertoire courant au path ────────────────────────────────────

sys.path.insert(0, os.path.dirname(**file**))

from flask import Flask, request, jsonify from storage_backend import StorageBackend from utils import count_tokens, sha256_text, ensure_dir

═══

CONFIGURATION

═══

CONFIG: Dict[str, Any] = { # Taille des réservoirs (en tokens) "reservoirs": { "past": 20000, "future": 10000, "present": 5000, }, # Auto-scale : agrandit le réservoir quand rempli à X % "scale_threshold": 0.70, "scale_factor": 1.5,

```
# Nombre de copies redondantes (recommandé : 4)
"doublon_copies": 4,

# Répertoires
"archive_dir": "./archives",
"data_dir":    "./data",
```

```
# Profils d'entité (seuils de pertinence pour le routage)
"entity_profiles": {
    "human": {"pass_threshold": 0.5,  "future_threshold": 0.7},
    "ia":    {"pass_threshold": 0.6,  "future_threshold": 0.7},
    "iah":   {"pass_threshold": 0.55, "future_threshold": 0.75},
    "agent": {"pass_threshold": 0.6,  "future_threshold": 0.65},
    "other": {"pass_threshold": 0.5,  "future_threshold": 0.7},
},

# Clé AES-256 (32 bytes). En production : utiliser un vault.
"aes_key": os.environ.get("AES_KEY", "skyclaw_default_key_32bytes_____"),

# Détection de perte de contexte
"context_loss_threshold": 0.25,   # alerte si baisse > 25 %
"context_loss_backups":   2,       # 2 sauvegardes automatiques sur alerte
```

}

ensure_dir(CONFIG["archive_dir"]) ensure_dir(CONFIG["data_dir"])

═══

ÉTAT GLOBAL (thread-safe)

═══

counters_lock = threading.Lock() counters: Dict[str, int] = {"past": 0, "future": 0, "present": 0} daily_ingest: Dict[str, int] = {"tokens": 0}

═══

INITIALISATION

═══

app = Flask(**name**) storage = StorageBackend(base_dir=CONFIG["data_dir"], doublon_copies=CONFIG["doublon_copies"], archive_dir=CONFIG["archive_dir"],)

LOGIQUE MÉTIER

def make_entry(interlocutor: str, partition: str, block: int, text: str, tags: list, relevance: float,) -> Dict[str, Any]: """Construit un enregistrement normalisé.""" ts = datetime.now(timezone.utc).isoformat() return { "id": sha256_text(f"{ts}|{interlocutor}|{text}"), "timestamp": ts, "interlocutor": interlocutor, "partition": partition, "block": block, "text": text, "tags": tags or [], "relevance_score": round(float(relevance), 4), "token_count": count_tokens(text), "agent": "SkyClaw-v2.0", }

def route_entry(entity_type: str, source: str, text: str, tags: list, relevance: float,) -> Dict[str, Any]: """ Routage intelligent vers la partition adaptée.

```
Règles :
- source='user'       → past  (mémoire des interactions humaines)
- source='assistant'  → selon relevance (past/future/present)
- source='other'      → selon relevance + profil entité
"""
profile = CONFIG["entity_profiles"].get(
    entity_type, CONFIG["entity_profiles"]["other"]
)

# Détermination partition & bloc
if source == "user":
    partition, block = "past", 2
elif source == "assistant":
    if relevance >= profile["pass_threshold"]:
        partition, block = "past", 1
    elif relevance >= profile["future_threshold"]:
        partition, block = "future", 1
    else:
        partition, block = "present", 1
else:
    if relevance >= profile["future_threshold"]:
        partition, block = "future", 2
    elif relevance >= profile["pass_threshold"]:
        partition, block = "past", 2
    else:
        partition, block = "present", 1

entry = make_entry(source, partition, block, text, tags, relevance)

# Déduplication
if storage.is_duplicate(entry["id"]):
    return {"status": "duplicate", "id": entry["id"]}

# Stockage
storage.store_entry(partition, entry)

# Mise à jour compteurs
```

```
with counters_lock:
    counters[partition]  += entry["token_count"]
    daily_ingest["tokens"] += entry["token_count"]

# Auto-scale si nécessaire
_maybe_autoscale(partition)

return {"status": "stored", "id": entry["id"], "partition": partition, "block":
block}
```

def _maybe_autoscale(partition: str) -> None: """Agrandit le réservoir si le taux de remplissage dépasse le seuil.""" with counters_lock: used = counters[partition] reserve = CONFIG["reservoirs"][partition]

```
if used >= CONFIG["scale_threshold"] * reserve:
    new_reserve = int(reserve * CONFIG["scale_factor"])
    storage.snapshot(partition)
    with counters_lock:
        CONFIG["reservoirs"][partition] = new_reserve
    msg = f"[autoscale] {partition} : {reserve} → {new_reserve} tokens"
    storage.add_alert(msg)
    print(msg)
```

def _emergency_backup() -> Dict[str, Any]: """ Déclenche 2 sauvegardes d'urgence (détection perte de contexte). Conforme au protocole JSON AVANT PERTE DE CONTEXTE. """ results = [] for i in range(CONFIG["context_loss_backups"]): try: n = storage.create_daily_doublons() results.append({"backup": i + 1, "status": "ok", "copies": n}) except Exception as e: results.append({"backup": i + 1, "status": "error", "detail": str(e)}) storage.add_alert(f"[URGENCE] Perte de contexte détectée — {len(results)} sauvegardes créées") return {"emergency_backups": results}

═══

ENDPOINTS REST

═══

@app.route("/memory/store", methods=["POST"]) def api_store(): """ Stocke un fragment de mémoire.

```
Body JSON :
{
    "entity_type": "human|ia|iah|agent|other",
    "source":      "user|assistant|other",
    "text":        "...",
    "tags":        ["tag1", "tag2"],     (optionnel)
    "relevance_score": 0.0-1.0           (optionnel, défaut 0.0)
}
"""
```

```
data = request.json or {}
required = ["entity_type", "source", "text"]
missing = [k for k in required if k not in data]
if missing:
    return jsonify({"error": "champs manquants", "missing": missing}), 400

result = route_entry(
    entity_type=data["entity_type"],
    source=data["source"],
    text=data["text"],
    tags=data.get("tags", []),
    relevance=float(data.get("relevance_score", 0.0)),
)
return jsonify(result), 200
```

@app.route("/memory/status", methods=["GET"]) def api_status(): """Retourne l'état complet du système mémoire.""" with counters_lock: status = { "agent": "SkyClaw-v2.0", "timestamp": datetime.now(timezone.utc).isoformat(), "tokens_used": counters.copy(), "reservoirs": CONFIG["reservoirs"].copy(), "daily_ingest": daily_ingest.copy(), "fill_rates": { p: round(counters[p] / CONFIG["reservoirs"][p], 3) for p in ("past", "future", "present") }, "alerts": storage.get_alerts(), "hologram_preview": storage.relecture_hologram() [:5], } return jsonify(status), 200

@app.route("/memory/restore", methods=["POST"]) def api_restore(): """ Restaure une partition depuis une sauvegarde.

```
Body JSON :
{
    "partition": "past|future|present",
    "method":    "doublon|zip",          (défaut: doublon)
    "index":     0-3                      (défaut: 0, pour doublon)
}
"""
data = request.json or {}
partition = data.get("partition")
if not partition:
    return jsonify({"error": "partition requise"}), 400
method = data.get("method", "doublon")
index  = int(data.get("index", 0))
try:
    path = storage.restore(partition, method=method, index=index)
    return jsonify({"status": "restored", "partition": partition, "path":
path}), 200
except Exception as e:
    return jsonify({"error": str(e)}), 500
```

@app.route("/memory/archive", methods=["POST"]) def api_archive(): """Crée une archive ZIP chiffrée + 4 copies redondantes.""" try: enc_path = storage.create_zip_archive(CONFIG["aes_key"]) n_copies = storage.create_daily_doublons() return jsonify({ "status": "archived", "archive": enc_path, "doublons": n_copies, "copies": CONFIG["doublon_copies"], }), 200 except Exception as e: return jsonify({"error": str(e)}), 500

@app.route("/memory/emergency_backup", methods=["POST"]) def api_emergency(): """ Déclenche 2 sauvegardes d'urgence. À appeler dès détection d'une perte de contexte. """ result = _emergency_backup() return jsonify(result), 200

@app.route("/memory/query", methods=["GET"]) def api_query(): """ Retourne les derniers enregistrements d'une partition. Params : ?partition=past&limit=20 """ partition = request.args.get("partition", "past") limit = int(request.args.get("limit", 20)) entries = storage.get_entries(partition, limit=limit) return jsonify({"partition": partition, "count": len(entries), "entries": entries}), 200

@app.route("/health", methods=["GET"]) def health(): """Vérification de l'état du serveur.""" return jsonify({"status": "ok", "agent": "SkyClaw-v2.0"}), 200

═══

DÉMARRAGE

═══

if **name** == "**main**": port = int(os.environ.get("PORT", 8080)) print(f"[SkyClaw Memory] Démarrage sur http://0.0.0.0:{port}") aes_msg = "(variable d env)" if os.environ.get('AES_KEY') else "(defaut CHANGER EN PROD)" print(f"[SkyClaw Memory] AES_KEY : {aes_msg}") app.run(host="0.0.0.0", port=port

storage_backend.py

storage_backend.py — SkyClaw Memory System

Python 3.10+ | Stockage JSONL + archives ZIP chiffrées + 4 sauvegardes

Auteur : SkyClaw (Skywork AI) | Projet : Eden Memory

import os import json import shutil import zipfile import threading from datetime import datetime, timezone from typing import Any, Dict, List, Optional

from utils import ensure_dir, write_jsonl, read_jsonl, sha256_text, encrypt_file_aes256, decrypt_file_aes256

class StorageBackend: """ Backend de stockage pour la mémoire SkyClaw.

```
Architecture :
 ./data/past/     — mémoire passée (blocs 1 & 2)
 ./data/future/   — mémoire future / projections
 ./data/present/  — contexte actuel
 ./archives/      — sauvegardes ZIP chiffrées
 ./archives/doublons/  — 4 copies redondantes
"""

PARTITIONS = ("past", "future", "present")

def __init__(self, base_dir: str, doublon_copies: int, archive_dir: str):
 self.base_dir = base_dir
 self.doublon_copies = max(1, doublon_copies)   # minimum 1, recommandé 4
 self.archive_dir = archive_dir
 self._lock = threading.Lock()
 self.alerts: List[str] = []

    # Création des répertoires
    ensure_dir(self.base_dir)
    ensure_dir(self.archive_dir)
    for p in self.PARTITIONS:
        ensure_dir(os.path.join(self.base_dir, p))

# ────────────────────────────────────────────────
# ÉCRITURE
# ────────────────────────────────────────────────

def store_entry(self, partition: str, entry: Dict[str, Any]) -> None:
    """Stocke un enregistrement dans la partition donnée."""
    if partition not in self.PARTITIONS:
        raise ValueError(f"Partition inconnue : {partition}")
    date = datetime.now(timezone.utc).date().isoformat()
    path = os.path.join(self.base_dir, partition, f"{date}.jsonl")
    with self._lock:
        write_jsonl(path, entry)

# ────────────────────────────────────────────────
# LECTURE & DÉDUPLICATION
# ────────────────────────────────────────────────

def is_duplicate(self, entry_id: str) -> bool:
    """Vérifie si un entry_id existe déjà dans toutes les partitions."""
    for p in self.PARTITIONS:
        pdir = os.path.join(self.base_dir, p)
        if not os.path.isdir(pdir):
            continue
        for fname in os.listdir(pdir):
            if fname.endswith(".jsonl"):
                for e in read_jsonl(os.path.join(pdir, fname)):
                    if e.get("id") == entry_id:
                        return True
    return False

def get_entries(self, partition: str, limit: int = 100) -> List[Dict[str, Any]]:
    """Récupère les N derniers enregistrements d'une partition."""
    pdir = os.path.join(self.base_dir, partition)
    if not os.path.isdir(pdir):
```

```
        return []
    entries = []
    for fname in sorted(os.listdir(pdir), reverse=True):
        if fname.endswith(".jsonl"):
            for e in read_jsonl(os.path.join(pdir, fname)):
                entries.append(e)
                if len(entries) >= limit:
                    return entries
    return entries

# ────────────────────────────────────────────────
# SAUVEGARDES (4 copies redondantes)
# ────────────────────────────────────────────────

def create_daily_doublons(self) -> int:
    """
    Crée jusqu'à 4 copies redondantes des fichiers du jour.
    Retourne le nombre de copies créées.
    """
    date = datetime.now(timezone.utc).date().isoformat()
    created = 0
    for p in self.PARTITIONS:
        src = os.path.join(self.base_dir, p, f"{date}.jsonl")
        if not os.path.exists(src):
            continue
        for i in range(self.doublon_copies):
            dst_dir = os.path.join(self.archive_dir, "doublons", p, f"copy_{i}")
            ensure_dir(dst_dir)
            dst = os.path.join(dst_dir, f"{date}.jsonl")
            shutil.copy2(src, dst)
            created += 1
    return created

def create_zip_archive(self, aes_key: str) -> str:
    """
    Crée une archive ZIP chiffrée AES-256 de toutes les partitions du jour.
    Retourne le chemin du fichier .zip.enc
    """
    date = datetime.now(timezone.utc).date().isoformat()
    zip_path = os.path.join(self.archive_dir, f"{date}_skyclaw_archive.zip")

    with zipfile.ZipFile(zip_path, "w", zipfile.ZIP_DEFLATED) as zf:
        for p in self.PARTITIONS:
            pdir = os.path.join(self.base_dir, p)
            if not os.path.isdir(pdir):
                continue
            for fname in os.listdir(pdir):
                if fname.endswith(".jsonl"):
                    full = os.path.join(pdir, fname)
                    zf.write(full, arcname=os.path.join(p, fname))

    enc_path = encrypt_file_aes256(zip_path, aes_key)
    os.remove(zip_path)
    return enc_path

def snapshot(self, partition: str) -> str:
    """
    Prend un snapshot horodaté d'une partition.
    Utilisé automatiquement lors d'un auto-scale.
    """
    ts = datetime.now(timezone.utc).strftime("%Y%m%dT%H%M%SZ")
    src = os.path.join(self.base_dir, partition)
```

```
    dst = os.path.join(self.archive_dir, f"{partition}_snapshot_{ts}")
    if os.path.isdir(src):
        shutil.copytree(src, dst)
    return dst

# ────────────────────────────────────────────────
# RESTAURATION
# ────────────────────────────────────────────────

def restore(self, partition: str, method: str = "doublon", index: int = 0) ->
str:
    """
    Restaure une partition depuis une sauvegarde.
    method : 'doublon' | 'zip'
    index  : numéro de la copie (0-3) pour method='doublon'
    """
    date = datetime.now(timezone.utc).date().isoformat()

    if method == "doublon":
        src = os.path.join(
            self.archive_dir, "doublons", partition, f"copy_{index}",
f"{date}.jsonl"
        )
        if not os.path.exists(src):
            raise FileNotFoundError(f"Doublon copy_{index} non trouvé pour
{partition}/{date}")
        dst = os.path.join(self.base_dir, partition, f"{date}_restored.jsonl")
        shutil.copy2(src, dst)
        return dst

    elif method == "zip":
        enc_files = sorted(
            [f for f in os.listdir(self.archive_dir) if
f.endswith("_skyclaw_archive.zip.enc")],
            reverse=True,
        )
        if not enc_files:
            raise FileNotFoundError("Aucune archive chiffrée trouvée")
        enc_path = os.path.join(self.archive_dir, enc_files[0])
        aes_key = os.environ.get("AES_KEY", "skyclaw_default_key_32bytes_____")
        dec_path = decrypt_file_aes256(enc_path, aes_key)
        with zipfile.ZipFile(dec_path, "r") as zf:
            members = [m for m in zf.namelist() if m.startswith(partition +
"/")]
            if not members:
                raise FileNotFoundError(f"Partition '{partition}' absente de
l'archive")
            for m in members:
                zf.extract(m, self.base_dir)
        os.remove(dec_path)
        return self.base_dir

    else:
        raise ValueError(f"Méthode inconnue : {method}. Utiliser 'doublon' ou
'zip'.")

# ────────────────────────────────────────────────
# MONITORING
# ────────────────────────────────────────────────

def add_alert(self, msg: str) -> None:
    ts = datetime.now(timezone.utc).isoformat()
```

```
    self.alerts.append({"timestamp": ts, "message": msg})

def get_alerts(self) -> List[Dict]:
    return list(self.alerts)

def relecture_hologram(self) -> List[Dict]:
    """
    Retourne les enregistrements à haute pertinence (score > 0.8)
    pour relecture prioritaire.
    """
    priorities = []
    for p in ("future", "past"):
        for e in self.get_entries(p, limit=500):
            if e.get("relevance_score", 0.0) > 0.8:
                priorities.append({
                    "id": e["id"],
                    "partition": p,
                    "text_preview": e.get("text", "")[:80],
                    "reason": "high_relevance",
                })
    return priorities

def count_tokens_total(self) -> Dict[str, int]:
    """Compte les tokens stockés par partition."""
    totals = {p: 0 for p in self.PARTITIONS}
    for p in self.PARTITIONS:
        for e in self.get_entries(p, limit=10000):
            totals[p] += e.get("token_count", 0)
    return totals
```

utils.py

utils.py — SkyClaw Memory System

Python 3.10+ | stdlib + pycryptodome

Auteur : SkyClaw (Skywork AI) | Projet : Eden Memory

import os import json import hashlib from pathlib import Path from typing import Any, Dict, Generator

---

---

1. FILESYSTEM

---

def ensure_dir(path: str) -> None: """Crée le répertoire (et ses parents) s'il n'existe pas.""" Path(path).mkdir(parents=True, exist_ok=True)

---

2. HACHAGE & TOKENS

---

def sha256_text(text: str) -> str: """Retourne le SHA-256 hex d'une chaîne UTF-8.""" return hashlib.sha256(text.encode("utf-8")).hexdigest()

def count_tokens(text: str) -> int: """ Compteur de tokens léger (split whitespace). Remplacer par tiktoken pour la précision en production. """ return max(1, len(text.split()))

---

3. JSONL (append-friendly)

---

def write_jsonl(path: str, entry: Dict[str, Any]) -> None: """Ajoute un enregistrement JSON à un fichier .jsonl.""" ensure_dir(os.path.dirname(path) or ".") with open(path, "a", encoding="utf-8") as f: f.write(json.dumps(entry, ensure_ascii=False) + "\n")

def read_jsonl(path: str) -> Generator[Dict[str, Any], None, None]: """Lit un fichier .jsonl ligne par ligne (générateur).""" if not os.path.exists(path): return with open(path, "r", encoding="utf-8") as f: for line in f: line = line.strip() if line: try: yield json.loads(line) except json.JSONDecodeError: continue

---

4. CHIFFREMENT AES-256-CBC (pycryptodome)

---

def _pad(data: bytes) -> bytes: pad_len = 16 - (len(data) % 16) return data + bytes([pad_len]) * pad_len

def _unpad(data: bytes) -> bytes: pad_len = data[-1] return data[:-pad_len]

def encrypt_file_aes256(path_in: str, aes_key: str) -> str: """ Chiffre un fichier en AES-256-CBC. Retourne le chemin du fichier chiffré (.enc). La clé doit faire exactement 32 caractères ASCII. """ from Crypto.Cipher import AES from Crypto.Random import get_random_bytes

```
key = aes_key.encode("utf-8")[:32].ljust(32, b"\x00")
iv = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_CBC, iv)

with open(path_in, "rb") as f:
    data = _pad(f.read())

out_path = path_in + ".enc"
with open(out_path, "wb") as fo:
    fo.write(iv + cipher.encrypt(data))
return out_path
```

def decrypt_file_aes256(path_in: str, aes_key: str) -> str: """ Déchiffre un fichier AES-256-CBC. Retourne le chemin du fichier déchiffré (.dec). """ from Crypto.Cipher import AES

```
key = aes_key.encode("utf-8")[:32].ljust(32, b"\x00")
with open(path_in, "rb") as f:
    iv = f.read(16)
    enc = f.read()

cipher = AES.new(key, AES.MODE_CBC, iv)
data = _unpad(cipher.decrypt(enc))

out_path = path_in + ".dec"
with open(out_path, "wb") as fo:
    fo.write(data)
return out_path
```

FIN ID-Y4